CAPTools Project: Evaluation and Application of the Computer Aided Parallelisation Tools

FINAL REPORT

David O'Neal
National Center for Supercomputing Applications
University of Illinois, Champaign, IL

Richard Luczak
University of Tennessee, Knoxville
Aeronautical Systems Center, Wright-Patterson Air Force Base, OH

Michael White
Ohio Aerospace Institute
Air Force Research Laboratory, Wright-Patterson Air Force Base, OH

Abstract

This report extends a previous study[1] of the Computer Aided Parallelisation Tools package (http://captools.gre.ac.uk/) developed by the University of Greenwich. Product effectiveness, deployment and training requirements, and the more general issue of continued support for the project, are all addressed in this installment.

Scalability and the level of effort required to achieve it are considered first. An informative inventory of essential features and basic usage strategies follows directly. A review of project accomplishments leads into a discussion of appropriate courses of action and then key recommendations are provided in closing.

[1] D. O'Neal, R. Luczak and M. White, *CAPTools Project: Evaluation and Application of the Computer Aided Parallelisation Tools*, Proceedings of the 1999 DoD HPC Users Group Conference, Monterey, CA.

Overview

During the past year, we have continued to work with the University of Greenwich on a software evaluation project involving their CAPTools product. ASC PET funding for the CAPTools work was awarded to the University of Illinois and the University of Tennessee in May of 1999, but negotiations with Greenwich stalled at an inopportune time and as a result, no funding was provided to UG for contract year 4 (CY4). However, UG was able to maintain a basic level of support through attenuation of CY3 funds. It's likely that awards will be provided to all of these players for CY5.

The origins of the ASC PET CAPTools Project were examined in our first paper[1]. This publication may be viewed at the NCSA CSM Group website.

http://www.ncsa.uiuc.edu/EP/CSM/publications/1999/UCG99_CAPTools.pdf

Unless noted to the contrary, all references to the CAPTools product correspond to a single integrated IRIX installation consisting of the latest CAPO executable furnished by NASA Ames and the message passing libraries, scripting tools, and manuals from the standard University of Greenwich package. All timing tests were performed on the Origin clusters at NCSA and ASC.

We begin this final installment with a brief review of Amdahl's Law for Parallelization. This section serves two purposes. It lends definition to basic terms used throughout the article while establishing a method for evaluating the effectiveness of CAPTools-generated parallel source codes. Notes describing the basic care and feeding of the software (Usage) lead into a discussion of more severe limitations (Caveats). Fundamental descriptions of each evaluation code precede more detailed observations regarding scalability (Applications). Significant findings are then summarized and final recommendations are made in closing.

Amdahl's Law of Parallelization

The premise of Amdahl's Law is that every algorithm has a sequential part that ultimately limits the speedup that can be achieved by a multiprocessor implementation. In this context, *speedup* is the ratio of execution times for single processor (dividend) and multiple processor (divisor) runs. The classical form of Amdahl's Law[2] is usually stated as follows:

> *If the serial component of an algorithm accounts for* $1/S$ *of its execution time, then the maximum speedup that can be attained by running it on a parallel computer is* S*.*

More detailed analyses generally begin with a characterization of a program solely in terms of its operation count. It is assumed that some portion q of these operations can be executed in parallel by p processors and that the remaining operations must be executed sequentially.

For a given problem size, if the total operation count and per-processor performance of the code are presumed constant for any value of *p*, then a simple expression for speedup may be written as a function of *p* and *q*.

$$S\,(p,q) = (q/p + 1 - q)^{-1}$$

This is Amdahl's Law of Parallelization. It is important to note that *S* reflects an *idealized* speedup value, also known as *false* speedup because parallel executions are known to involve overheads that invalidate the aforementioned presumptions. Nevertheless, this function does serve as an upper bound for more realistic speedup models that incorporate operation counts and performance levels that are dependent on the processor count[3].

Error! Not a valid link.

Figure 1. Amdahl's Law of Parallelization

Estimated speedups derived from timing measurements for each of our evaluation codes are used to estimate the portion of the total operation count parallelized by CAPTools. Reference is made to the curves of Fig. 1. Note that in order to make the interesting part of the speedup curves easier to interpret, values for $q < 90\%$ have been cropped from the chart. Speedups associated with the missing portions of these curves are considered to be unacceptably poor and in some cases are only reported as such without accompanying data. Reference is also made to the *efficiency* of a parallel code. This value is taken to be the ratio of speedup (dividend) to the processor count (divisor), or $e = S/p$. A novel interpretation of Amdahl's Law of Parallelization developed by O'Neal and Urbanic showed that these concepts can also be used to estimate the performance of cache-based microprocessors on the basis of memory component bandwidths.[4]

Usage Notes

This section describes limitations and general usage characteristics associated with the current 2.1 Beta version of the CAPTools product. Both new and experienced users should find it informative. Note that at the time of this writing, the release of version 2.2 Beta was imminent.

CAPTools supports standard Fortran 77 syntax (an F90 parser is in development). Fortran statements that deviate from F77 standard may need to be rewritten. An unusual I/O format and a number of instances of array syntax were encountered and (justifiably) rejected by CAPTools during testing.

Source files containing conditional compilation directives must be preprocessed prior to loading. The CAPTools parser treats compiler directives as comments and therefore <u>all</u> program statements are loaded. Unless an error occurs as a side effect of unwanted inclusions, no warning is provided to the user.

A new feature called the Undefined Symbol Browser has recently been added. In the past, all external references had to be resolved before a source code could be loaded successfully. For example, stubs corresponding to library calls had to be provided by the user, and then these same stub routines would have to be commented back out of the CAPTools-generated source code prior to compilation.

The new window supports examination and control of undefined symbols and their effect on subsequent dependency analyses. Help is currently not available for this feature. If the default assumptions are accepted, the Analyser window is automatically raised. This is also the case when a source without undefined symbols has been successfully loaded. A better choice might be to raise the READ Knowledge editor as its use is indicated first.

User and READ statement knowledge should be added prior to initiating a dependency analysis. Integer logic associated with variables appearing in READ statements can be simplified merely by indicating the sign of these inputs. More complex logic can also be developed. Of course it may not be possible to make such determinations for all variables, but in general, user knowledge should be specified whenever possible.

After successfully loading a source code, the Analyser button becomes active. Two distinct algorithms for determining dependencies are supported (Banerjee, Omega). Any combination of three dependency test options (Scalar, Exact, Disproofs) may be performed within any combination of three contexts (Interprocedural, Knowledge, Logic). A group of preset selections is also provided (Basic, Intermediate, Full). Ultimately a full analysis should be performed, but when working with large sources, a progressive approach is recommended. Performance of the Analyser may be improved by (1) saving the database, (2) exiting and re-entering the system, and (3) reloading the database file after completing each stage.

The dependency analysis kernel presumes static allocation of all variables (as if SAVE statements were implemented everywhere). The use of this programming technique is often found in older programs. Static allocation implies that the addresses of local variables are fixed, thus when a subroutine (function) is exited, its local variables persist. Therefore, all calls that might be executed more than once will spawn dependencies. If static allocation is not presumed, such dependencies should be identified and removed.

Configurable windows are provided for filtering and editing dependencies. The pruning of false dependencies might be regarded as an option, but the scalability of the final code can be greatly affected by the presence of such dependencies. The function of the Directives Browser (OpenMP) and the Dependency Graph Viewer (message passing) should be well understood before electing to forego their use. Unfortunately, as of this writing the only guidance provided for any of the features developed by NASA Ames appears only in the form of a README file.

OpenMP source may be generated immediately after completing any dependency analysis, but the use of the Directives Browser is highly recommended. After loading a source code and completing an analysis, users should proceed directly to the Directives Browser. Dependencies inhibiting loop parallelization should be viewed and edited there if possible. Partitioning is not required.

Message passing codes must be partitioned and communication logic must also be developed within the environment before a new source code can be generated. The compilation and execution of CAPTools-generated message passing codes is somewhat more demanding of the user, but the difference is approximately the same as that between compiling and running any given OpenMP source and its MPI equivalent.

CAPTools generated message passing code contains calls to various library routines developed by UG. At the core of this design lies CAPLib, a library of communication routines built upon various message passing primitives (MPI, PVM, shmem). This wrapper-like layer was implemented before MPI[5] had emerged as a de facto standard interface. A nice paper describing CAPLib was recently made available at the CAPTools website.[6]

Caveats

Consideration of the following commandment for CAPTools users is of utmost importance:

Thou shalt not save a database after generating source code

In the current version of CAPTools, the contents of the database are corrupted during the logic dump associated with code generation. Although it is possible to continue to working within a session after generating source code, it is not recommended. Oddly enough, this rather severe shortcoming has remained undocumented for at least six months.

Another insidious bug originates from the so-called Openwin libraries that are an integral part of the CAPTools environment. After successfully starting and then exiting a CAPTools session, subsequent attempts to start new sessions unexpectedly terminate during initialization. The most reliable workaround for the problem is to touch the libxview.a and libolgx.a library files, but of course only the owner of the files can apply it.

Because of the aforementioned database corruption problem, users are forced to exit and re-enter CAPTools after generating source code. Then the Openwin problem will often prevent immediate re-entry. Not an ideal situation where new users are involved. UG has been working on a new build of the windowing libraries and a File menu feature that supports multiple database loads. We expect these items will soon be resolved.

Applications

Basic characteristics of the simulation codes selected for evaluation are followed by detailed descriptions of the individual porting efforts and scalability measurements taken with the final versions of the new parallel application codes are featured herein.

We begin this section with a table of information summarizing each of our evaluation codes in terms of its geometry, subroutine and function count, number of statements before and after application of CAPTools, the cases (partition sizes) for which results have been validated.

Application	Model Geometry	Subroutines and Functions	F77 Lines In	OMP Lines Out	MPI Lines Out	OMP Validation	MPI Validation
R-Jet	3D	63	7655	7471	19284	1,2,4,8,16,32	1,2,4
FDL3DI	3D	76	9964	15330	11993	1-8,16,32,64	1-8,16
N-Body	3D	2	195	207	376	1-6	1-6
PFEM	2D	16	2073	1934	2357	1-8,16,32,64	1-8,16,32,64

Table 1. Summary of general information regarding CAPTools test applications

The first two applications (R-Jet and FDL3DI) were provided by AFRL. The others (N-Body and PFEM) are research codes out of Rice University and Carnegie Mellon University respectively.

In the following sections, brief descriptions of the functionality of each application are followed by a set of detailed notes regarding the porting effort. Speedup curves are provided for each experiment.

R-Jet
Air Force Research Laboratory

R-Jet is a hybrid, high-order, compact finite difference spectral method for simulating vortex dynamics and breakdown in turbulent jets. While the code is explicit in time, the compact finite difference scheme requires inversion of tridiagonal matrices, giving rise to the same sort of parallelization problems exhibited by implicit methods.

The R-Jet program was only recently implemented as a serial code and so a parallel version had not yet been started when CAPTools became available. It was, however, designed to support the solution of radial and axial derivatives by either an explicit differencing scheme or the compact method described above, which ultimately proved to have a profound effect on scalability.

R-Jet employs a programming technique in which complex valued arrays are mapped onto pairs of real arrays for subsequent input to FFT routines. This approach was problematic for earlier versions of CAPTools. Dummy routines were needed to maintain the geometry of the data structures. Although CAPTools was later amended to eliminate the underlying restrictions, we continued to maintain the new code because its use resulted in a significant reduction of the time required to complete a dependency analysis.

CAPTools was overly cautious in its assessment of the utilization of a number of work arrays. The pruning of a number of false dependencies was performed with the Dependency Graph browser (see Usage Notes section). One unexpected result was the placement of message passing statements outside of existing IF-THEN constructs whose use was required for proper masking of data exchanges. For some of our test problems, this resulted in the generation of spurious messages. It was fixed by applying the aforementioned IF-THEN statements to the message passing calls.

The most significant deficiency we observed with respect to the R-Jet code was the way in which the collection of summary information needed to generate the final solution and restart files was implemented. The code generated by CAPTools represented a sort of *bucket brigade* arrangement in which variables were handed down one at a time from processor to processor until they reached PE0 and were written out to disk. For larger partition sizes, problems with default values for MPI environment settings were observed.

We chose to replace this code with a simpler and more efficient implementation in which each processor was programmed to open a uniquely named temporary file into which local data was subsequently streamed using unformatted block writes. The master processor (PE0) was then used to gather and construct the global solution and restart files by post-processing the array of temporary files. The new method outperformed the CAPTools-generated code for all cases involving more than a few processors where it still remained competitive.

Error! Not a valid link.

Figure 2. R-Jet Speedup Data (101x131x17)

The MPI curves presented in Fig. 2 illustrate scalability of the resultant message passing code through 64 processors for a 101x131x17 test case partitioned in the first two dimensions. Speedups for the compact differencing scheme level off at around 25 processors due to the serial dependence implied by the tridiagonals, but the efficiency of the explicit differencing algorithm exceeds 50% through full range of partition sizes tested ($p = n^2$ for n ranging from 1 to 8). The estimated fraction of the total operation count parallelized by CAPTools exceeded 97% for the latter case.

Reference is also made to a set of OpenMP experiments performed last fall with an earlier version of CAPTools 2.1 Beta and the compact differencing source. Results were presented by Greenwich at Supercomputing 1999. The data for the OpenMP (OMP) curve was taken from a chart that appeared on the reverse side of the handout distributed by Greenwich at SC'99:

http://www.gre.ac.uk/~lp01/sc99/openmp/page2.html

The OMP curve indicates that between 92% and 95% of the total operation count was parallelized by CAPTools.

FDL3DI
Air Force Research Laboratory

The Flight Dynamics Laboratory 3D Implicit code is used to study aeroelastic effects. It was developed by the AFRL Basic CFD Research Group and Wright State University. FDL3DI is a high-frequency Navier-Stokes model featuring a one-dimensional structural solver component. Collective use of the numerous variations of FDL3DI maintained by AFRL research staff is heavy. Interest in developing a recipe for converting all of these sources to parallel form is what led to the inclusion of the FDL3DI code in this study.

A parallel implementation based on the use of a Chimera overset grid decomposition scheme was modified by Wright State University to run in a serial fashion. The original design produced by K. Tomko (WSU) was quite clever, making full use of the knowledge that FDL3DI was already configured to handle overset grids. By characterizing arbitrary mesh decompositions as overset grid problems, the task was reduced to a geometry problem. A new tool capable of imposing such decompositions had to be developed from scratch, but very few changes to the original code were required. This is an excellent example of the difference between the ways that humans and machines work.

Error! Not a valid link.

Figure 3. FDL3DI Speedup Data (100x100x100)

The serialized WSU code was used as input to CAPTools. Both OpenMP and message passing sources were produced. The Directives Browser showed that all but a few loops were automatically parallelized without any additional work, but as indicated by the speedup curves of Fig. 3, the resulting code was only able to keep a few processors busy. However, the effort required to achieve this improvement was insignificant.

Work on a new message passing version of FDL3DI was less satisfactory. After completing a full analysis, an array index was selected for partitioning. CAPTools uses this information to create a sort of template for application to other, similarly dimensioned program arrays including those of lower rank. Upon completion of this phase, maskings for all of the newly partitioned arrays were computed and the corresponding communications statements were generated. Attempts to apply the overlapping communications and memory reduction features resulted in segmentation faults.

Tests of the message passing executables created from 1- and 2-dimensional partitionings of the data were unsuccessful. Iteration timings that consumed about a minute of CPU while running on a single processor required over 2 minutes when 2 processors were used, and over 4 minutes when 4 processors were used. The need to increase one of the default per-processor limits, i.e. MPI_MSGS_PER_PROC, gave indication that a large number of small messages were being dumped onto the network. Unnecessary data replication meant that the physical memories of each participating processor were also being oversubscribed. All available network topologies were tested. Sources generated from default and pruned dependency graphs were checked. The pipe configuration coupled with the source produced from the pruned graph resulted in the best timings (noted above). The trend was clear and so testing was suspended at this point.

Reference is also made to another set of experiments with a newer, modified version of the FDL3DI code conducted by members of the Ohio Supercomputer Center research staff. Results were presented by the University of Greenwich at Supercomputing 1999 in the form of two handouts. Charts appeared on the reverse side of these documents.

http://www.gre.ac.uk/~lp01/sc99/openmp/page2.html
http://www.gre.ac.uk/~lp01/sc99/mp/page2.html

The OSC tests were carried out with a more recent version of the FDL3DI source featuring additional memory allocations in the form of specialized workspace arrays. There is also some indication that they were solving a different (larger) problem. However, the dependency graph pruning techniques applied in both cases were essentially the same, which means that the tuning effort given to each of these codes was also approximately the same. Note, however, that the results of the OSC tests were much more positive than ours.

Timing data presented for the OSC OpenMP tests suggests that about 90% of the total operation count was parallelized by CAPTools. Speedup data for the message passing runs indicates that 96% to 98% of the total operation count was parallelized by CAPTools. The significance of this observation is it implies that subtle differences in source files can have a dramatic effect on the efficiency of the source code generated by CAPTools.

N-Body
Rice University

N-Body is a three-dimensional n-body model with constraints. It is used to solve problems arising in electromagnetic applications. It determines equilibrium positions for an arbitrary set of points interacting on the surface of a unit sphere according to a $1/d^2$ force law (here d denotes the distance between two points). The result is a particular uniform distribution of points in three dimensions.

The serial version of the N-Body code was used as input to CAPTools. A hand-written MPI code was also developed for comparison purposes. The dependency analysis and the generation of the OpenMP code were accomplished in a fully automatic way. Partitioning phases associated with the message passing model required a minimum of user input. Both CAPTools-generated codes produced correct output.

Identical initial distributions and convergence criteria were used for all experiments. Measurements were taken for problems consisting of 100 and 1,000 points running on various partition sizes. Approximately 10^5 and 10^6 iterations respectively were required to achieve equilibrium. Because N-Body doesn't involve a computational mesh, a full topology was necessarily specified. Positional updates for each particle depend only on the governing ordinary differential equation.

Error! Not a valid link.

Figure 4. N-Body Speedup Chart

Scalability measurements for the OpenMP and handwritten MPI versions of the code are presented in Fig. 4. Results associated with the OpenMP model were excellent. Good scalability was observed through 16 processors for the smaller problem and through 64 processors for the larger one. The estimated portion of the total operation counts parallelized by CAPTools ranged from 90% to 99%.

A few runs of the CAPTools-generated message passing code were also performed for various partition sizes. Although correct results were produced, we were unable to achieve an acceptable level of scalability for either problem size.

The N-Body tests revealed a specific limitation of the current CAPTools message passing model. Simple communication patterns may be implemented in overly general fashions. In one case, we found that data arrays were being transmitted a single element at a time inside of a loop. In another, a reduction operation wasn't recognized as such. It had also been rewritten in terms of CAPLib primitives, e.g. CAP_SEND and CAP_RECEIVE, where a collective call was indicated.

These items imply that hand-tuning of message passing source codes may be required in order to realize acceptable levels of performance. Where the communication of data arrays is involved, at least some of these changes may be completely straightforward and very effective, but collective operations are generally more complicated. Knowledge of message passing concepts is prerequisite.

We should also mention that the memory reduction feature associated with the message passing model was successfully applied. Note that if memory reduction is <u>not</u> applied, all data arrays will be replicated across all processor memories. While data replication might be acceptable early in a design cycle, it doesn't represent a valid approach for a final product.

PFEM
Carnegie Mellon University

The Parallel Finite Element Method code is a research-level finite element code suitable for solving highly nonlinear boundary value problems in two dimensions. The use of nonconforming finite element geometries facilitates a simple domain decomposition strategy based on two-colorings. Each subdomain consists of approximately half of the total number of elements. All element calculations within each subdomain are effectively uncoupled[7].

The PFEM model was selected for evaluation for a number of reasons. A hand-coded directive-driven source code based on the Cray Research microtasking model (CRI) was available for comparison purposes. PFEM also featured a complete set of validation problems and timer outputs that saved us a bit of time and effort. We were also interested in testing CAPTools' ability to deal with dynamic allocation of arrays, a feature that figured prominently in PFEM's design.

The CRI parallel source was used as input to CAPTools. Existing directives were converted to CAP comments by the parser. Unresolved references to a couple of system functions (SECOND and GETENV) were encountered and processed by the new Undefined Symbol Browser. User knowledge reflecting a positive element count and tolerance value was added, and then a full analysis was performed.

A few circular (self-dependent) references associated with the static treatment of local variables were pruned from the dependency graph (see Usage Notes section), the database was saved (see Caveats section), and then just prior to exiting the program, a new OpenMP source code was generated. Following a simple makefile change (the addition of a compiler switch) we were able to build and run a new OpenMP program. Work on the message passing model, however, proved to be much more complicated by comparison.

Upon restarting CAPTools, the database that had been saved prior to generating the OpenMP source was loaded, giving us a bit of a head start on the message passing version. Partitioning was specified with respect to the first index of a single array, the geometry of which was representative of all of the primary data arrays. An unstructured mesh was then selected and partitioning was initiated. Within the context of CAPTools, an unstructured mesh corresponds to a fully connected messaging topology. In general, heavy use of indirect addressing is indicative of the presence of an unstructured mesh.

After confirming array bounds for a short list of references, all of the primary data arrays were automatically partitioned. A few indexing arrays were necessarily deleted from the list of partitioned arrays. Indeed, when this step was omitted, the resultant executable produced incorrect output. Masking and communication calculations were completed using the default system settings and the final database was saved and a new message passing source code was created just prior to exiting CAPTools.

Our attentions then turned to the set of *cap* utility scripts, e.g. *capmake* and *capf90*, that were developed to support proper compilation and linking of CAPTools-generated message passing codes. We quickly discovered that documentation describing the construction of executables was quite thin, and so we chose to design a makefile around the *capf90* script. A minor problem was observed while attempting to pass options to the underlying compiler, but a suitable workaround was found and we were able to complete the build. The result was a single executable that could be configured at run time to accommodate any problem or partition size.

Follow-up experiments with the available message passing optimization buttons were also carried out. Enabling the use of asynchronous calls (overlapping communications) produced no changes in the output source code. Attempts to apply the memory reduction feature resulted in segmentation faults, but we later discovered that this feature is not supported for unstructured mesh parallelization in the current release and therefore the button should be inactive. See the Release Notes page for CAPTools version 2.0 Beta dated October 23, 1998, for further detail.

Error! Not a valid link.

Figure 5. PFEM Speedup Chart

Initial timings were surprising. The message passing version was quite a bit more efficient than the corresponding OpenMP code, particularly for larger partition sizes. Estimated fractions of the total operation count parallelized by CAPTools were in the range of 95% to 97% for the message passing model, but only around 90% for the OpenMP code. Because the PFEM source code was originally written for execution on a multiprocessor computer, we had expected more from the OpenMP port. This led us to take a closer look at the source with the Directives Browser.

Cursory inspection revealed that a pair of scattering loops had not been parallelized because of the presence of indirect addressing arrays. CAPTools could not possibly have determined that these scattering statements were actually permutations and that no true dependencies were present, so the inhibitors were explicitly removed using the Directives Browser and a new source was generated. The desired effect was then implemented automatically. Scalability of the new OpenMP executable was much improved. The estimated fraction of total operations parallelized by CAPTools was now exceptional, ranging from 95% to 98%. These results are reflected by the OpenMP curves shown in Fig. 5.

Summary

The CAPTools product was evaluated with respect to its ability to deal with two significant DoD codes and two academic research programs. A total of 15 cases were considered including 3 produced by the Ohio Supercomputer Center. At least some measure of speedup was observed for all 7 of the CAPTools-generated OpenMP codes while only 5 of the 8 CAPLib experiments met with similar results. The effort required to achieve these results varied from case to case, but the OpenMP tests were consistently less demanding both in terms of parallel programming requirements and time.

The memory reduction feature could not be applied to one of the test codes (PFEM) due to the presence of an unstructured mesh. In another case (FDL3DI), attempts to apply the feature resulted in segmentation faults. When memory reduction is not (or cannot be) applied, the entire dataset is replicated.

The CAPLib model requires much more from the user in every way. Detailed knowledge of the input source code is presumed. Data distribution must be considered prior to any other parallelization steps. A thorough understanding of parallel programming concepts is required in order to guide CAPTools through the process. CAPLib source code is also much more difficult to debug, even for the experienced programmer. The absence of a compact and detailed CAPLib document describing the application interface contributed to the problem.

Unwanted modification of original formatting contained by input source codes renders output from the *diff* utility useless. Nearly all statements are affected. As a result, nontrivial changes can only be observed through side-by-side comparison of entire programs.

The effectiveness of the CAPLib model is highly dependent on the presence of a well-defined mesh. Our work with the N-Body code made this perfectly clear. Conversely, CAPTools-generated OpenMP code is not subject to the same sort of restriction. Results for the OpenMP version of N-Body were truly exceptional.

For two of our test cases, CAPTools implemented block-oriented communications of data arrays as loops around the arrays in which elements were being transmitted one element at a time. This limitation is reportedly slated for improvement in a future release. Until then, users may need to modify output source code “by hand” in order to achieve acceptable levels of performance.

We also observed that in some cases, CAPTools did not recognize situations that called for the use of collective communication calls. Instead, less efficient code was written in terms of CAPLib primitives. This particular problem is much more difficult to deal with, but hopefully it will also be resolved in the near term. Otherwise, hand tuning may be required as indicated by the last line of the preceding paragraph.

Recommendations

Users faced with the challenge of parallelizing a FORTRAN 77 code should certainly consider application of the CAPTools package. Work may be required to successfully load or analyze any given input file, but once accomplished, the value of the information represented in the CAPTools Call Graph, the Dependency Graph and the Directives Browser is significant. The effort required to achieve this minimal level of progress is considered worthwhile for any and all porting projects.

Where the option exists, users are advised to work with the OpenMP model. Compilers supporting OpenMP are usually associated with distributed shared memory platforms and so we are indirectly recommending that users target DSM machines like the SGI Origin 2000 first. Source code can be generated immediately after completing an analysis. If formatting changes are neglected, the output file looks very much like the input file, thus reducing the impact of any subsequent debugging or tuning efforts. The build process is also completely straightforward. The simplicity of this approach is very attractive. For some of our test cases, the OpenMP executables were also the most effective in terms of scalability. As previously noted, the OpenMP model is currently the only practical choice for codes that do not involve a mesh.

The OpenMP executable out of Ames (version 2.1 Beta) has been a robust and practical tool for many months now. We recommend broad deployment of this application across all of the HPCMP computing centers.

We are less enthusiastic about the CAPTools message passing model. It is most effective in the hands of an experienced programmer, but this type of DoD user is not inclined to develop message passing logic that depends on proprietary libraries. Novice programmers may find the CAPLib application interface less complicated than MPI for example, but such users are not well suited to the demands imposed by the model, especially when debugging or tuning is required (as is often the case). The former situation isn't likely to change, and the possibility of a member of the latter group producing a significant result within a reasonable amount of time is remote.

Therefore, we recommend that funding for the development of CAPTools features that support the OpenMP model should be continued. Deliverables associated with the current version of the UG proposal for CY5 should be revised if necessary. In addition to the directives-based features, other components the might be improved include the dependency analysis kernel, the windowing libraries, the file manager, and the help file system. Specific items of interest include:

- Core support
- Updated Openwin libraries
- Enhanced database management features
- Elimination of unnecessary source formatting changes
- Improved documentation and help file system

Anything that is intrinsically bound to the message passing model should be neglected. Additional support for the development of a web-based instructional presentation focusing on the CAPTools OpenMP model should also be considered.

The collaboration between UG and NASA Ames has been very successful, but we would still like to see a single product emerge. Otherwise, the future of CAPTools becomes much murkier. Anything less than a truly integrated effort may not be sustainable within the narrow market that is addressed by CAPTools.

Acknowledgments

We wish to express our sincere thanks to Professor Mark Cross, Drs. Constantinos Ierotheou and Steven Johnson, and Peter Leggett of the Parallel Processing Research Group, University of Greenwich, London, England, UK, and to Dr. Henry Jin of the NASA Ames Research Center, Moffet Field, CA, without whose help this project would not have been possible.

We would also like to thank Drs. Miguel Visbal and Raymond Gordnier (AFRL) and Professor Karen Tomko (Wright State University) for their contributions to the project including the provision of the FDL3DI application.

References

1. D. O'Neal, R. Luczak, and M. White, *CAPTools Project: Evaluation and Application of the Computer Aided Parallelisation Tools*, proceedings of the DoD High Performance Computing Users Group Conference, Monterrey, CA, 1999.
2. G. Amdahl, *Validity of the single-processor approach to achieving large-scale computational capabilities*, proceedings of the AFIPS Conference, volume 30, page 483, AFIPS Press, 1967.
3. W. Schönauer, *Scientific Supercomputing: Architecture and Use of Shared and Distributed Memory Parallel Computers*, self-edition, Karlsruhe, Germany, 2000.
4. D. O’Neal and J. Urbanic, *On Microprocessors, Memory Hierarchies, and Amdahl’s Law*, proceedings of the DoD High Performance Computing Users Group Conference, Monterrey, CA, 1999.
5. *MPI: A Message Passing Interface Standard*, University of Tennessee, Knoxville, TN, May 5, 1994.
6. P. Leggett, S. Johnson, and M. Cross, *CAPLib: A Thin Layer Message Passing Library to Support Computational Mechanics Codes on Distributed Memory Systems*, internal report, Parallel Processing Research Group, Center for Numerical Modelling and Process Analysis, University of Greenwich, London, UK, 2000.
7. D. O'Neal and R. Reddy, *The Parallel Finite Element Method*, in proceedings of the Cray User Group Inc., Spring Conference, Denver, CO, 1995.
8. *OpenMP Fortran Application Program Interface*, Version 1.0, October, 1997.
9. *Computer Aided Parallelization Tools User's Guide*, Parallel Processing Research Group, University of Greenwich, London, UK, Version 2.0 Beta, October, 1998.